PyGen: A MATLAB/Simulink Based Tool for Synthesizing Parameterized and Energy Efficient Designs Using FPGAs

Jingzhao Ou and Viktor K. Prasanna
Department of Electrical Engineering, University of Southern California
Los Angeles, California, 90089-2560 USA
Email: {ouj,prasanna}@usc.edu

Abstract

System level tools based on MATLAB/Simulink are becoming popular for designing applications using FPGAs. In these tools, application designers describe their designs at high level using the powerful modeling environment provided by MATLAB/Simulink. Then, these designs are automatically translated into corresponding FPGA implementations. However, there is a lack of support for developing parameterized and energy efficient designs using these tools. In this paper, we propose PyGen*, an add-on tool, to address this issue. The four major functionalities offered by our tool are: development of parameterized designs; integration of a domain-specific modeling technique for rapid and accurate energy estimation; profile of energy dissipation and feedback to application designers; flexible interface for design space traversal and identification of energy efficient designs. To illustrate the design process using the tool and to show its effectiveness, details of designs for an FFT kernel and an adaptive beamforming application are shown. For the adaptive beamforming application, the identified design achieves up to 30% energy reduction compared with other designs considered in our experiments.*

I. Introduction

Increasing density and integration of pre-compiled hardware cores, such as embedded multipliers, memory blocks, and RISC processors, etc., have made FPGAs an attractive option for implementing complex signal processing applications. A recent trend towards application specific FPGAs, which allows the creation of FPGAs with a mix of hardware resources and optimizes their performance for a specific application area [19], adds to such attractiveness. Traditionally, the performance metrics for implementing many embedded systems have been latency and throughput. With the proliferation of portable and mobile devices, energy efficiency has also become an important performance metric. One example is software-defined radio (SDR). In SDR, dissimilar and complex wireless standards (e.g. GSM, IS-95, *cdma2000*) are processed in a single adaptive base station. On-the-fly processing of large amount of data from mobile terminals demands high computational requirements. State-of-the-art RISC processors and DSPs are unable to meet such high processing requirements of the base stations. Minimizing the energy dissipation of these base stations has also become an issue. This is because the base stations are usually wireless and distributed and thus work in an energy constrained environment. FPGAs stand out as an attractive option for implementing various functions of SDR due to their high performance, low power dissipation per computation, and reconfigurability [6].

As FPGAs are being used to implement many complex signal processing algorithms, describing FPGA designs using hardware description languages (HDLs) is too time consuming and unattractive. It can be a bottleneck in the communication between the hardware designer and the algorithm developer as application designers in the signal processing community are usually not familiar with HDLs. MATLAB/Simulink based design tools, such as *DSP Builder* [2] from Altera and *System Generator* [22] from Xilinx, are becoming popular and have been shown to be capable of bridging this gap for developing signal processing applications. *System Generator* has a block set through which the designer can get access to proprietary IP cores and HDL designs. Application designers assemble designs by dragging and dropping the blocks from the block set to their designs and connecting them via a GUI. The block set contains (1) blocks that represent the basic hardware resources such as registers, multiplexers, etc.; (2) blocks that represent control logic, mathematical functions, and memory; (3) blocks that represent proprietary IP cores such as FFT, DCT, etc. Besides, there is a *Resource Estimator* block that allows application designers to quickly estimate the resource utilization of their designs.

There are several advantages offered by these MATLAB/Simulink based design tools. One is that there is no need to know HDLs. This allows researchers and users

from the signal processing community, who are usually familiar with the MATLAB/Simulink modeling environment, to get involved in the hardware design process. Another advantage is that the designer can make use of the powerful modeling environment offered by MATLAB/Simulink to perform arithmetic level simulation, which is much faster than behavioral and architectural simulations in traditional FPGA design flows [12].

However, there are also some limitations using the current MATLAB/Simulink design flow to optimize the energy performance of the applications. The current tools have no support for *rapid* energy estimation for FPGA designs. One reason is that energy estimation using RTL (Register Transfer Level) simulation (which can be accurate) is too time consuming and can be overwhelming considering the fact that there are usually many possible implementations of an application on FPGAs. The other reason is that the basic elements of FPGAs are look-up tables (LUTs), which are too low-level an entity to be considered for high level modeling and rapid energy estimation. No single high level model can capture the energy dissipation behavior of all possible implementations on FPGAs. A rapid energy estimation technique based on domain-specific modeling is presented in [3] and is shown to be capable of quickly obtaining fairly accurate estimate of energy dissipation of FPGA designs. However, we are not aware of any tools that integrate such rapid energy estimation techniques.

Another limitation is that these tools do not provide interface for describing design constraints, traversing the MATLAB/Simulink design space, and identifying energy efficient FPGA implementations. Therefore, while algorithms such as the ones proposed in [17] are able to identify energy efficient designs for reconfigurable architectures, they cannot be directly integrated into the current MATLAB/Simulink based design tools.

To address the above limitations, we develop *PyGen*, an add-on tool that provides additional functionalities to the available MATLAB/Simulink based design tools. It is written in Python scripting language [18]. By creating an interface between Python and the MATLAB/Simulink based system level design tools, our tool allows the use of Python language for describing FPGA designs in MATLAB/Simulink. This provides several benefits. First, it enables the development of *parameterized* designs. Parameters related to application requirements (e.g. data precision) and those related to hardware implementations (e.g. hardware binding) can be captured by *PyGen* designs. It also enables *rapid* and *accurate* energy estimation by integrating a domain-specific modeling technique and using the switching activity information from MATLAB/Simulink simulation. Finally, it makes the identification of energy efficient designs possible by providing a flexible interface to traverse the design space.

The paper is organized as follows. Section II discusses related work. Section III describes the software architecture and design flow of *PyGen*. Due to its wide availability, we focus on enhancing *System Generator* for developing parameterized and energy efficient designs. However, by making some changes, our tool can be used for other MATLAB/Simulink based design tools. Application design using *PyGen* is divided into two levels. Kernel level development is discussed in Section IV-A while application level development is discussed in Section IV-B. To illustrate the design process using our tool, details of designs for an FFT kernel and an adaptive beamforming application using the proposed design tool are shown in Section V. We conclude in Section VI.

II. Related Work

jg is a tool developed by Xilinx [20], which uses Java to describe *System Generator* designs. In *jg*, application designers compile their Java code, execute it to generate intermediate MATLAB program, which can be used to generate *System Generator* designs. Compared with *jg*, we use Python, instead of Java, to describe the designs. Since Python is a scripting language, its clear and concise syntax makes such description easier than Java. Most importantly, the current version of *jg* has no support for rapid energy estimation and system level optimization.

There are system level design tools such as DK2 [5] from Celoxica and Forge [7] from Xilinx which use high-level languages such as C and Java for FPGA designs. When using these tools, the application designers describe their applications using C or Java and rely on the compiler to infer the appropriate architecture for implementing the application and to perform optimizations such as loop unrolling, pipelining, etc. The output of these tools is either HDL code or EDIF netlist. A C-to-VHDL high-level synthesis framework is proposed in [8]. The input to their framework is C code and they employ a set of compiler transformations to optimize the synthesized designs. None of these tools address synthesis of energy efficient designs.

Taking an approach entirely different from those taken by the Java and C based tools discussed above, MATLAB/Simulink based design tools provide a high level abstraction of the underlying hardware resources and allow the application designers to describe the data flow and its hardware realization directly through this high level abstraction. *PyGen* manipulates this high level abstraction using Python scripting language.

In our experiments, we noticed that MATLAB/Simulink based tools produce designs with better performance than other system level design tools in many cases. This is because generic HDL description is usually not enough to achieve best performance as the recent FPGAs integrate many heterogeneous components. Use of device specific design constraint files and vendor IP cores as that in the MATLAB/Simulink based design flow plays an important

role in achieving good performance. To illustrate this, we consider three implementations of 18×18-bit multiplication on Virtex-II Pro FPGAs using the embedded multipliers. In the first implementation, only VHDL is used to describe the design. In the second implementation, timing constraints are added to the HDL description to optimize the timing performance of the design. In the third implementation, we describe the design using *System Generator*, which is an MATLAB/Simulink based design tool from Xilinx, and use the IP core for multiplication. The maximum operating frequency F_{max} of these implementations is shown in Table I. The design that uses IP core has by far the highest maximum operating frequency. Since energy dissipation depends on operating frequency, such differences will have a significant impact on energy efficiency as well. The reason for such performance improvement is that the specific locations of the embedded multipliers require appropriate connections between the multipliers and the registers around them. Use of appropriate location and timing constraints as in the generation of the IP cores leads to improved performance when using these multipliers [1]. Since *PyGen* is built upon MATLAB/Simulink, we expect that the designs using it can result in this superior timing and energy performance as well.

TABLE I. Maximum operating frequency of various implementations of 18×18-bit multiplication

Implementation	VHDL	VHDL with timing constraints	*System Generator* with IP cores
$F_{\max}$	120 MHz	207 MHz	354 MHz

III. Software Architecture

PyGen is written in Python, which is an object-oriented scripting language with concise syntax, flexible data types and dynamic typing [18]. It is widely used in many software systems. There are also attempts to use Python for hardware designs [13].

The software architecture of *PyGen* is shown in Figure 1. It contains four major modules. The architecture and the function of these modules are described in the following.

A. *PyGen* Module

The *PyGen* module is a Python module. It is responsible for creating communication between *PyGen* and MATLAB/Simulink and mapping the basic building blocks in *System Generator* to Python classes.

MATLAB provides three ways for creating such communication: MATLAB COM (Component Object Model) server, MATLAB engine, and a Java interface [14]. We build the communication interface through the MATLAB

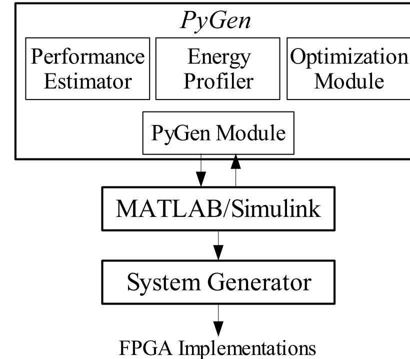


Fig. 1. Architecture of *PyGen*

COM server by using the Python Win32 extensions from [9]. Through this interface, *PyGen* and *System Generator* can obtain the relevant information from each other and control the behavior of each other. For example, moving a design block in *System Generator* can change the placement properties of the corresponding Python object and vice versa. After a design is described in Python, the *PyGen* module communicates with MATLAB/Simulink and creates a corresponding design in Simulink. Since the *PyGen* module is a basic module, application designers are required to import it first using the script `import PyGen` every time they describe their designs in Python.

Using some specific naming convention, the *PyGen* module maps the basic block set provided by *System Generator* to the corresponding classes (*basic classes*) in Python, which is shown in Figure 2. For example, block *xbsBasic_r3/Mux*, which is a *System Generator* block representing hardware multiplexers, is mapped to a Python class `CxlMul`. All the design parameters of this block, such as *inputs* (number of inputs), *precision* (precision), are mapped to the data attributes of its corresponding class and are accessible as `CxlMul.inputs` and `CxlMul.precision`. The information on the input and output ports of the blocks is stored in data attribute *ips* and *ops*. Therefore, for two Python objects *A* and *B*, *A.ips[0:2] = B.ops[2:4]* has the same effect as connecting the third and fourth output ports of block *B* to the first two input ports of *A*.

Using the *PyGen* module, application designers describe their designs by instantiating classes from the Python class library, which is equivalent to dragging and dropping blocks from the *System Generator* block set to their designs. By leveraging the object-oriented class inheritance in Python, application designers can extend the class library by creating their own classes (*extended classes*, represented by the shaded blocks in Figure 2) and derive parameterized designs. This is further discussed in Section IV-A.1.

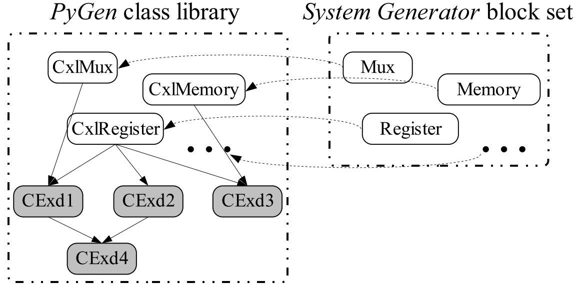


Fig. 2. Python class library within *PyGen*

B.Performance Estimator

After a *PyGen* class is instantiated, there is a performance model associated with the generated object. The performance model captures the performance of this object, such as resource utilization, latency, and energy dissipation, etc. The resource utilization can be obtained by invoking the *Resource Estimator* block provided by *System Generator* and parsing its output. We are currently interested in the numbers of slices, the amount of Block RAM, and the number of embedded multipliers used in a design. Regarding latency, it can be obtained directly from the *latency* data attribute if the object is instantiated from the basic classes, or can be calculated based on the construction and the data attributes of the object if the object is instantiated from the extended classes. To obtain the energy performance, we integrate a domain-specific modeling technique for rapid and accurate energy estimation proposed in [3]. This is further discussed in Section IV-A.2.

C.Energy Profiler

The energy profiler can analyze the energy dissipation of a given component and interconnect of a design. The design flow using the profiler is shown in Figure 3. After the design is created, the application designers follow the standard FPGA design flow to synthesize and implement the design. Design files (.ncd files) that represent the FPGA netlist are generated. Then, it is simulated using ModelSim to generate simulation files (.vcd files). These files record the switching activity of the various hardware components on the device. The design files (.vcd files) and the simulation files (.vcd files) are then fed back to the profiler within *PyGen*. The profiler has an interface with XPower [22] and can obtain the average power consumption of the clock network, nets, logic, and I/O pads by querying XPower through this interface. Since the VHDL code generated by *System Generator* maintains the naming hierarchy of the original design, the profiler sums up these power values according to this naming hierarchy and outputs the power consumption of *System Generator* blocks or *PyGen* objects. Combining with appropriate timing information, the power values can be further translated to values of energy dissipation.

The energy profiler can identify the energy *hot spots* in designs. More importantly, as discussed in Section IV-B.3, it can be used to generate feedback information and to improve the accuracy of the performance estimator.

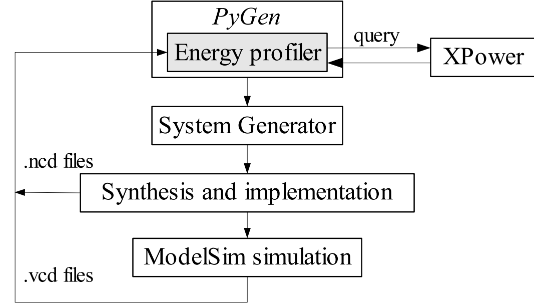


Fig. 3. Design flow using the energy profiler

D.Optimization Module

The optimization module provides two functions: description of the design constraints and optimization of the design with respect to the constraints. Since parameterized designs are developed as Python classes, application designers realize the two functions by writing Python code and manipulating the *PyGen* classes. This gives the designers complete flexibility to incorporate a variety of optimization algorithms and provides a way to quickly traverse the MATLAB/Simulink design space.

IV. Design Flow

Based on the architecture of *PyGen* discussed above, the design flow is illustrated in Figure 4. The shaded boxes represent the four major functionalities offered by *PyGen* in addition to the original MATLAB/Simulink design flow.

• *Parameterized design development.* Parameterized designs are described in Python. Design parameters such as data precision, degree of parallelism, hardware binding, etc., can be captured by the Python designs. After the designs are completed, *PyGen* is invoked to translate the designs in Python to the corresponding designs in MATLAB/Simulink. Changes to the MATLAB/Simulink designs, such as the adjustment of the placement of the blocks, also get reflected in the *PyGen* environment through the communication channel between them.

• *Performance estimation.* Using the modeling environment of MATLAB/Simulink, application designers can perform arithmetic level simulation to verify the correctness of their designs. Then, by providing the simulation results to the performance estimator within *PyGen* and

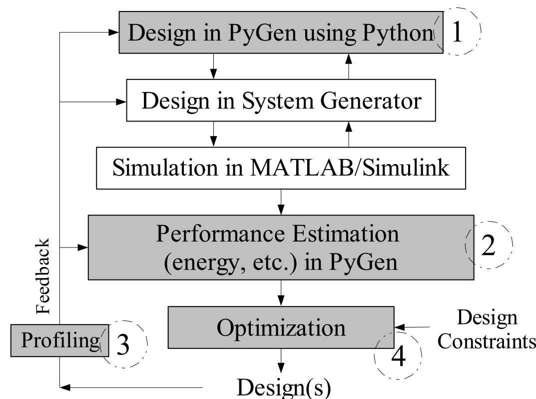


Fig. 4. Design flow of *PyGen*

invoking it, application designers can quickly estimate the performance of their designs, such as energy dissipation and resource utilization.

• *Optimization for energy efficiency.* Application designers provide design constraints, such as end-to-end latency, throughput, number of available slices and embedded multipliers, etc., to the optimization module. After optimization is completed, *PyGen* outputs the designs which have the maximum energy efficiency according to the performance metrics used while satisfying the design requirements.

• *Profile and feedback.* The design process can be iterative. Using the energy profiler, *PyGen* can break down the results from low-level simulation and profile energy dissipation of various components of the candidate designs. The application designers can use this profiling to adjust the architectures and algorithms used in their designs. Such energy profiling information can also be used to refine the energy estimates from the performance estimator.

Finally, using *System Generator* to generate the corresponding VHDL code, application designers can follow the standard FPGA design flow to synthesize and implement these designs and download them to the target devices.

The input to our design tool is a task graph. That is, the target application is decomposed into a set of tasks with communication between them. Then, the development using *PyGen* is divided into two levels: kernel level and application level. The objectives of kernel level development are to develop parameterized designs for each task and to provide support for rapid energy estimation. The objectives of application level development are to describe the application using the available kernels and to optimize its energy performance with respect to design constraints.

A.Kernel Level Development

The kernel level development consists of two design steps, which are discussed below.

1) Parametrized Kernel Development: As shown in [3], different implementations of a task (e.g. kernel) provides different design trade-offs for application development. Taking matrix multiplication as an example, designs with a lower degree of parallelism require less hardware resources than those with a higher degree of parallelism while introducing a larger latency. Also, at the implementation level, several trade-offs are available. For example, in the realization of storage, registers, slice-based RAMs and Block RAMs can be used. These implementations offer different energy efficiency depending on the size of data that needs to be stored. The objective of parameterized kernel design is to capture these design and implementation trade-offs and make them available for application development.

While *System Generator* offers limited support for developing parameterized kernels, *PyGen* has a systematic mechanism for this purpose by the way of Python classes. Application designers expand the Python class library and create *extended classes*. Each *extended class* is constructed as a tree, which contains a hierarchy of subclasses. The leaf nodes of the tree are *basic classes* while the other nodes are *extended classes*. An example of such an *extended class* is shown in Figure 5. This example illustrates some extended classes in the construction of a parameterized FFT kernel in *PyGen*. Once an *extended class* is instantiated, its subclasses also get instantiated. While translating this to the MATLAB/Simulink environment by the *PyGen* module, it has the same effect as generating *subsystems* in MATLAB/Simulink, dragging and dropping a number of blocks into these *subsystems*, and connecting the blocks and the *subsystems* according to the relationship between the classes.

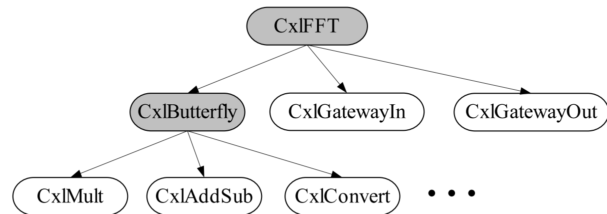


Fig. 5. Structure of the tree within the Python extended classes for parameterized FFT kernel development

Application designers are interested in some design parameters while generating the kernels. These parameters can be architecture used, hardware binding of a specific function, data precision, degree of parallelism, etc. We use the data attributes of the Python classes to capture these design parameters. Each design parameter of interest has a corresponding data attribute in the Python class. These data attributes control the behavior of the Python class when the class is instantiated to generate *System Generator* designs. They determine the blocks used in a MATLAB/Simulink design and the connections between the blocks. Besides, by

properly packaging the classes, the application designers can choose to expose only the data attributes of interest for application level development.

2) Support for Rapid and Accurate Energy Estimation: While the parameterized kernel development can potentially offer a large design space, being able to quickly and accurately obtain the performance of a given kernel is crucial for identifying the appropriate parameters of this kernel and optimize the performance of the application using it. To address this issue, we integrate into *PyGen* a domain-specific modeling based rapid energy estimation technique proposed in [3].

The use of domain-specific energy modeling for FPGAs is shown in Figure 6. In general, a kernel can be implemented using different architectures. For example, implementing matrix multiplication on FPGAs can employ a single processor or a systolic architecture. Implementations using a particular architecture are grouped into a domain. Analysis of energy dissipation of the kernel is performed within each domain. Because each domain corresponds to an architecture, energy functions can be derived for each domain. These functions are used for rapid energy estimation for implementations in the corresponding domain. See [3] for more details regarding domain-specific modeling.

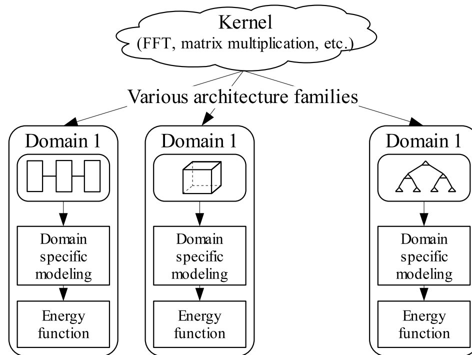


Fig. 6. Domain-specific modeling for rapid energy estimation

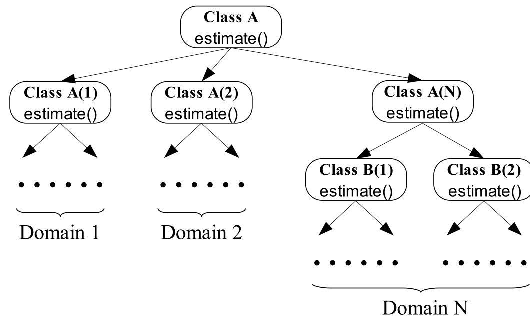


Fig. 7. Tree of classes organized as domains

In order to support this domain-specific modeling technique, the kernel developers must be able to group different kernel designs into the corresponding domain. Such support is not available in *System Generator* as the organization of the block set is fixed. However, after mapping the block set to the flexible class library in *PyGen*, re-organization of the class hierarchy according to the architectures represented by the classes becomes possible. Taking the case shown in Figure 7 as an example, Python class A represents various implementations of a kernel. It contains a number of subclasses $A(1)$, $A(2)$, $\cdots$, $A(N)$. Each of the subclasses represents the implementations of the kernel that belong to the same *domain*.

The process of energy estimation in *PyGen* is hierarchical. Energy functions are associated with the Python *basic classes* and are obtained through low-level simulation. They capture the energy performance of these basic classes under various possible parameter settings. For the *extended classes*, depending on whether domain-specific energy modeling is performed for the classes or not, there may be no energy functions associated with them for energy estimation. In case that the energy function is not available, energy estimate of the class needs to be obtained from the classes contained in it. While this way of estimation is fast by skipping the derivation of energy functions, it has lower estimation accuracy as shown in Table II in Section V-A.

To support such hierarchical estimation process, a method `estimate()` is associated with each Python object. When this method is invoked, it checks if an energy function is associated with the Python object. If yes, it calculates the energy dissipation of this object according to the energy function and the parameter settings for this object. Otherwise, *PyGen* iteratively searches the tree as shown in Figure 5 within this Python object until enough information is obtained to calculate the energy performance of the object. In the worst case, it will trace all the way back to the leaf nodes of the tree. Then, the `estimate()` method computes the energy performance of the Python object using these energy functions obtained as described above.

Switching activities within a design are a key factor that affects energy dissipation. By utilizing the data from MATLAB/Simulink simulation, *PyGen* obtains the *actual* switching activity of various blocks in the high level designs and uses them for energy estimation. Comparing with the approach in [3] which assumes default switching activities, this helps increase the accuracy of the estimates. To show the benefits offered by *PyGen*, we consider an 8-point FFT using the unfolded architecture discussed in Section V-A. It contains twelve butterflies, each based on the same architecture. In Figure 8, the bars show the power consumption of these butterflies while the upper curve shows the average switching activity of the *System Generator* basic building blocks used by each butterfly. Such switching activity information can be *quickly* obtained from the MATLAB/Simulink arithmetic level simulation. As shown in the figure, the switching activity information obtained from MATLAB/Simulink is

able to capture the variation of the power consumption of these butterflies. The average estimation error based on such switching activity information is 2.9%. For the sake of comparison, we perform energy estimation by assuming a default switching activity as in [3]. The results are shown in Figure 9. For default switching activities ranging from 20% to 40%, which are typical of designs of many signal processing applications, the average estimation errors can go up to as much as 36.5%. Thus, by utilizing the MATLAB/Simulink simulation results, *PyGen* improves the estimation accuracy.

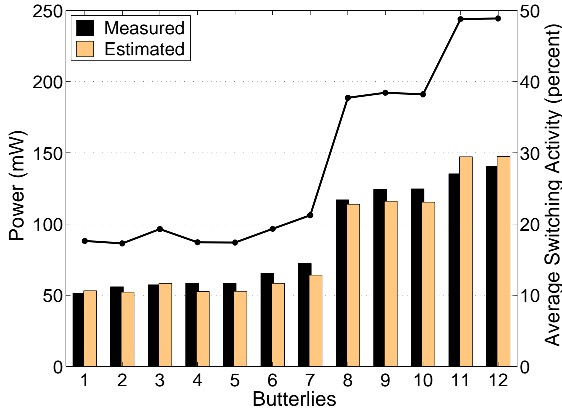


Fig. 8. Power consumption and average switching activities of input/output data of the butterflies in an unfolded-architecture for 8-point FFT computation

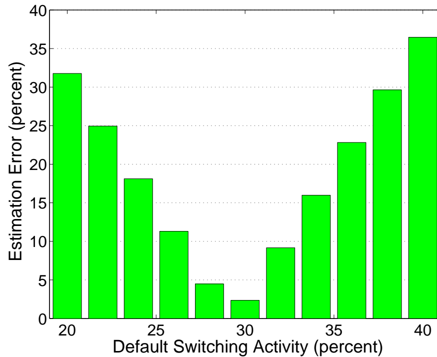


Fig. 9. Estimation error of the butterflies when default switching activity is used

B.Application Level Development

The application level development begins after the parameterized designs for the tasks are made available by going through the kernel level development. It consists of three design steps, which are discussed below.

1) Describing the Application: Based on the input task graph, the application designers construct the application using the parameterized kernels as discussed in the previous section. This is accomplished by manipulating the Python classes created in a way as described in Section IV-A.1. Besides, application designers need to create interfacing classes for describing the communication between tasks. These classes capture: (1) data buffering requirement between the tasks, which is determined by the application requirements and the data transmission patterns of the implementations of the tasks; (2) hardware binding of the buffering.

2) Support for Optimization: Application designers have complete flexibility in implementing the optimization module by handling the Python objects. For example, if the task graph of the application is a linear pipeline, the application designer can create a trellis as shown in Figure 10. Many signal processing applications including the beamforming application discussed in the next section can be described as linear pipelines. The parameterized kernel classes capture the various possible implementations of the tasks (the shaded circles on the trellis) while the interfacing classes capture the various possible ways of communication between the tasks (the connection between the shaded circles on the trellis). Then, the dynamic programing algorithm proposed in [17] can be applied to find out the design parameters for the tasks so that the energy dissipation of executing one data sample is minimized.

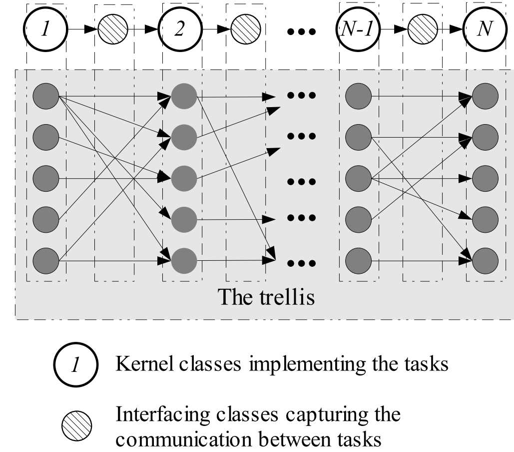


Fig. 10. Trellis for describing linear pipeline applications

3) Energy Profiling: By using the energy profiler in *PyGen*, application designers can write Python code to obtain the power or energy dissipation for a specific Python object or a specific kind of objects. For example, the power consumption of the butterflies used in an FFT design is shown in Figure 8. Based on the profiling, the application designers can identify the energy *hot spots* and change the designs of the kernels or the task graph of the applications to further increase energy efficiency of their designs. They can also use the profiling to refine the energy estimates from the energy estimator. One major reason that

necessitates such refinement is that the energy estimation using the energy functions (discussed in Section IV-A.2) captures the energy dissipation of the Python objects; it cannot capture the energy dissipated by the interconnect that provides communication between these objects.

V. Illustrative Examples

To illustrate the design process using *PyGen*, we present the development of an FFT kernel and an adaptive beamforming application that uses the kernel. The current version of *PyGen* is built upon *System Generator* 3.2. For the experiments discussed in the paper, we use Synplify Pro 7.2 [21] for synthesis, ISE 5.2.03 [22] for implementation, and ModelSim 5.7 [15] for simulation. Our target devices are Xilinx Virtex-II Pro series FPGAs. The measured resource utilization is obtained from the place-and-route report files (.par files) after implementing the designs using ISE. The measured power consumption is obtained by using the data from MATLAB/Simulink simulation to simulate the post place-and-route models in ModelSim. Besides, by analyzing the requirements of the software defined radio systems where the kernel and the application are widely employed, we set the operating frequencies of the designs at 200 MHz (except for the 16-point FFT design using an unfolded architecture, which operates at 135 MHz) and the data precision at 16 bits.

In our examples, *energy efficiency* is defined as the energy dissipation for processing one data sample. When we consider streaming data processing and assume that all the modules are active throughout the processing, energy efficiency can be measured as the average power consumption of the design. Also, *quiescent power*, which is the power consumption of the device when there is no switching activity on it, and the input/output power of the I/O pads are not considered since they are fixed once the target device and the input/output data requirements are determined. Their energy efficiency cannot be improved using our tool.

A. Kernel Level Development: Fast Fourier Transform

Fast Fourier Transform (FFT) is widely used in many signal and image processing applications. It is the key technique in OFDM (Orthogonal Frequency Domain Multiplexing) for significantly reducing the computation complexity. OFDM is being deployed in the realization of many high speed wireless LAN and ultra wideband (UWB) communication systems, such as the multiband OFDM systems proposed in [16]. Thus, energy efficient FFT designs are highly desired.

1) Development of Parameterized Kernel Designs: The parameterized FFT design is developed as a Python class `CxlFFT`. It contains two subclasses which use two different architectures for implementing the FFT kernel: unfolded and folded architecture. For the *unfolded architecture*, the FFT computation is flattened and spread out on the FPGA device. This architecture achieves the highest throughput and has little control and storage overhead. Thus, it is expected to have high energy efficiency. However, it also requires the largest amount of FPGA resources, which otherwise can be used for improving the energy efficiency of other tasks in the application. The *folded architecture* for FFT computation is shown in Figure 11. By repeatedly using the butterflies in the computation, it requires a much smaller amount of resource than that of the unfolded one. Also, the degree of parallelism can be varied. This provides design trade-offs in area and time.

Based on the application requirements as discussed in Section V-B, we identify the design parameters of interest and associate them with the corresponding data attributes of the Python class. These data attributes and the design parameters they represent are shown as below.

- *Frq*: operating frequency
- *nPnt*: number of frequency points
- *Arch*: architecture (unfolded or folded)
- *Sto*: hardware binding of storage elements (registers, slice-based RAM or Block RAM)
- *degPar*: degree of parallelism
- *Precision*: data precision

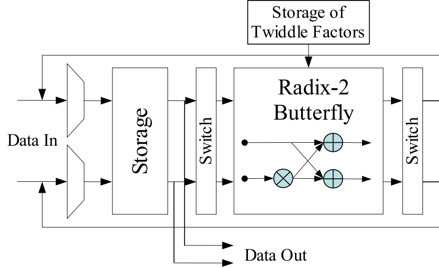
Multiplication within the butterfly is performed using embedded multipliers. Note that, in order to analyze the impact of switching activity on energy estimation, all the butterflies use the same architecture and multiplication with ± 1 and $\pm j$ is not bypassed in the design.

2) Performance Estimation: The performance of various instantiations of the `CxlFFT` class is shown in Table II. The estimated data are obtained through the *PyGen* performance estimator while the measured data are obtained through low-level simulation.

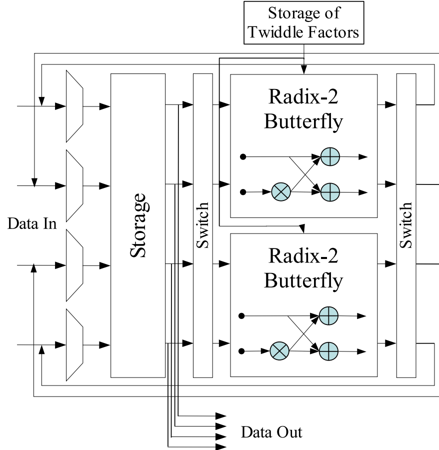
We perform a two-step power estimation for the FFT kernel. In the first step, since no energy function is associated with the derived classes, the *PyGen* performance estimator traces back the class hierarchy within the `CxlFFT` class and reaches the *basic classes*. The power estimation is obtained by summing up the power values of these *basic classes*. The values are shown in the row denoted as *Estimated (Step 1)*. In the second step, we analyze the performance of the FFT kernel by performing domain-specific modeling and deriving energy functions for each architecture employed by the kernel (*domain*). Such analysis can make use of the energy profiler. For example, using the profiling information as shown in Figure 12, we can estimate the communication costs among the building blocks within the butterflies. These costs cannot be captured when we analyze the energy performance of individual building blocks. The power estimates are computed using these energy functions. The

TABLE II. Power consumption and estimation errors of various implementations of FFT kernel

Design Parameters	*Arch*	Unfolded	Unfolded	Folded	Folded	Folded	Folded
	nPnt	8	16	16	16	16	16
	degPar	—	—	1	2	1	2
	Frq	200	135	200	200	200	200
	Sto	register	register	SRAM	SRAM	BRAM	BRAM
Power (*mW*)	Estimated (Step 1)	1278(12%)	2475(18%)	189(10%)	232(15%)	244(9%)	282(13%)
	Estimated (Step 2)	1379(5%)	2777(8%)	197(6%)	251(8%)	257(4%)	305(6%)
	Measured	1452	3018	210	273	268	324



(a) *degPar* = 1



(b) *degPar* = 2

Fig. 11. Folded architecture for FFT computation

values are shown in the row denoted as *Estimated (Step 2)*. Comparing with the measured data obtained through low-level simulation (the row denoted as *Measured*), we have estimation errors ranging from 9% to 18% for *Step 1*, and ranging from 4% to 8% for *Step 2*. On the average, 6% improvement in estmation accuracy is observed by going from *Step 1* to *Step 2*.

B. Application Level Development: MVDR Spectrum Calculation

In this section, we show the design of an MVDR (Minimum Variance Distortionless Response) spectrum calculation application [10] in order to illustrate the application level development using *PyGen*. This application is part of the MVDR adaptive beamforming process. Adaptive beamforming is used by many telecommunication systems

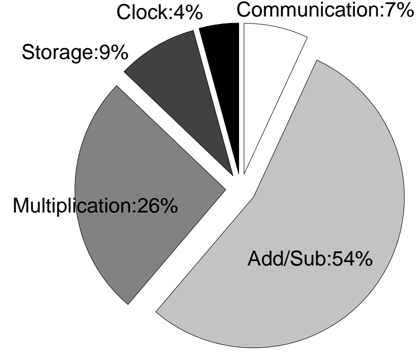


Fig. 12. Profile of the power consumption of the butterfly used in 8-point unfolded-architecture for FFT

such as software defined radio systems for better utilization of the limited radio spectrum. Energy efficiency is an important metric for implementing this application as these systems are usually battery operated.

The task graph of the application is shown in Figure 13, which consists of three tasks: Levinson Durbin recursion, correlation of the predictor coefficients, and spectrum calculation using FFT.

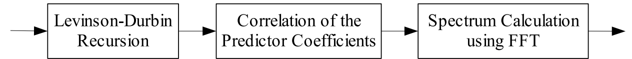


Fig. 13. Task graph of the MVDR application

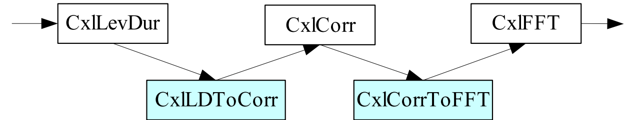


Fig. 14. Python classes for the MVDR application

1) Describing the Application: Following a similar process as shown in Section V-A, we develop kernel designs for the Levinson Durbin task (class `CxlLevDur`) and the correlation task (class `CxlCorr`). The design parameters captured by these two classes are: *Frq* (operating frequency), *M* (number of antenna elements in the system), *degPar* (degree of parallelism), and *Precision* (precision of the data). Two interfacing classes, `CxlLDToCorr` and `CxlCorrToFFT`, are also developed to describe the data communication between the tasks. The development of these classes is not included in this paper due to space

limitation. The relationships between the kernel classes and the interfacing classes are specified in Python and are illustrated in Figure 14. They represent different implementations of the MVDR application. Based on the application requirement in [4], we set M = 8, Frq = 200 (MHz), and $Precision$ = 16 (bit).

2) Optimization for Energy Efficiency: To illustrate the effectiveness of our tool, we perform an exhaustive search on various implementations of the MVDR application by instantiating the Python classes with different design parameters. We identify designs which have the minimum energy dissipation for processing one sample data based on our coarse estimates while ensuring that the designs of the complete application can fit into our target device (Xilinx Virtex-II Pro xc2vp20).

To show the effectiveness of *PyGen*, we identify five designs. They correspond to the five designs with lowest energy dissipation based on their measured energy performance. Figure 15 shows the measured and estimated energy performance of these designs. The coarse and refined estimations are based on the *Step 1* and *Step 2* estimation of the tasks. They are obtained as described in Section IV-A.2. The average estimation error for various designs of the MVDR application improves from 12% to 6% by using the refined estimates over the coarse estimates. After traversing the MATLAB/Simulink designs, the identified design (e.g. the left most design shown in Figure 15) can achieve an energy reduction of up to 30% compared with the designs considered.

3) Energy Profiling: The energy profiler can be used to obtain the energy dissipation of each task as shown in Figure 15. It can also be used to obtain the energy dissipation of various components within a task as shown in Figure 12. Such profiling helps to derive refined energy estimates for the tasks as discussed in Section V-A.2.

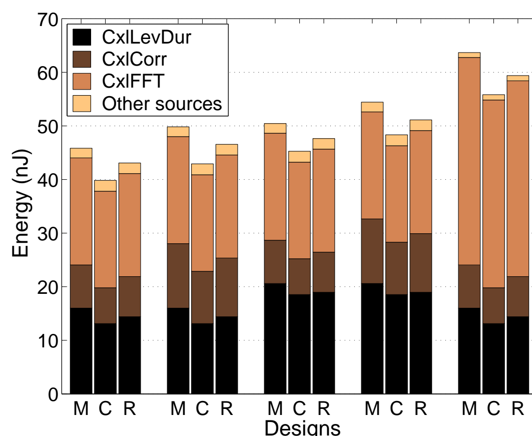


Fig. 15. Energy performance of various designs of the MVDR application (M denotes measured data; C denotes data from coarse estimation; R denotes data from refined estimation)

VI. Conclusion

A MATLAB/Simulink based design tool, *PyGen*, is presented in this paper. It can be used to develop parameterized and energy efficient FPGA designs using MATLAB/Simulink based system level design tools. We demonstrated the design flow and the effectiveness of the tool by providing two illustrative design examples.

The development of this tool is in progress. One issue is that the *System Generator* designs created from the Python code may have "strange" appearances in some cases since we use a simple algorithm for placing the blocks. Integration of sophisticated placement algorithms from open source projects such as [11] is required to resolve this issue. Finally, as FPGAs are integrating RISC processors, we expect that our tool can be further enhanced to develop energy efficient hardware/software designs.

References

[1] M. Adhiwiyogo, "Optimal Pipelining of I/O Ports of the Virtex-II Multiplier," *Xilinx Appli. Notes*, 2003.
[2] Altera, Inc., `http://www.altera.com`.
[3] S. Choi, J.-W. Jang, S. Mohanty, V. K. Prasanna, "Domain-Specific Modeling for Rapid System-Wide Energy Estimation of Reconfigurable Architectures," *Engr. of Reconf. Sys. & Algo. (ERSA)*, 2002.
[4] M. Devlin, "How to Make Smart Antenna Arrays," *Xilinx XCell Journal*, Issue 45, 2003.
[5] DK2, Celoxica, Inc., `http://www.celoxica.com/products/tools/dk.asp`.
[6] C. Dick, "The Platform FPGA: Enabling the Software Radio," *Software Defined Radio Tech. Conf. and Product Expo. (SDR)*, 2002.
[7] Forge, Xilinx, Inc., `http://www.xilinx.com/ise/advanced/forge.htm`.
[8] S. Gupta, M. Luthra, N. Dutt, R. Gupta, A. Nicolau, "Hardware and Interface Synthesis of FPGA Blocks using Parallelizing Code Transformations," *Parall. & Dist. Computing Sys. (PDCS)*, 2003.
[9] Mark Hammond, Python for Windows Extensions, `starship.python.net/crew/mhammond`.
[10] S. Haykin, "Adaptive Filter Theory," 3rd Edition, *Prentice Hall*, 1991.
[11] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting, "A CAD Suite for High-Performance FPGA Design," *Field Customizable Computing Machines (FCCM)*, 1999.
[12] J. Hwang, B. Milne, N. Shirazi, J. Stroomer, "System Level Tools for DSP in FPGAs," *Field Programmable Logic & Applications (FPL)*, 2001.
[13] P. Haglund, O. Mencer, W. Luk, B. Tai, "PyHDL: Hardware Scripting with Python," *Engr. of Reconf. Sys. & Algo. (ERSA)*, 2003.
[14] MathWorks, Inc., `www.mathworks.com`.
[15] Mentor Graphics, Inc., `www.mentor.com`.
[16] Multiband OFDM Alliance, `www.multibandofdm.org`.
[17] J. Ou, S. Choi, V. K. Prasanna, "Performance Modeling of Reconfigurable SoC Architectures and Energy-Efficient Mapping of a Class of Applications," *Field Customizable Computing Machines (FCCM)*, 2003.
[18] Python, `http://www.python.org`.
[19] C. Souza, "IP Columns Support Application-Specific FPGAs," *EE Times*, 2003.
[20] J. Stroomer, J. Ballagh, H. Ma, B. Milne, J. Hwang, N. Shirazi, "Creating System Generator Design Using *jg*," *Field Customizable Computing Machines (FCCM)*, 2003.
[21] Synplicity, Inc., `www.synplicity.com`.
[22] Xilinx Corporation, Inc., `www.xilinx.com`.